MIT Reversible Computing Project
Memo #M1*
"Time-Symmetric Control-Flow Instructions for Less Garbage in Reversible Programs"
DRAFT

Michael Frank
mpf@ai.mit.edu

Started 12/5/95
`Revision: 1.21`
December 19, 1996

Abstract

The most straightforward approach to implementing control-flow instructions (such as branches) reversibly is to push the source address for the branch onto a special stack [5]. Unfortunately this approach accumulates garbage whenever a branch is taken, garbage that cannot be disposed of under program control.

A potentially more space-efficient approach is to store the source address in a "branch register" which can be cleared under program control. However, the branch register approach seems clumsy and inelegant.

To solve these problems, we borrow from physics the concept of "time-symmetry," which says that the same sorts of operations should apply whether time (or in this case, program execution) is run forwards or backwards. Inspired by this notion, we can design time-symmetric branching, looping, and procedure-calling instructions that can be used to greatly reduce, and in many cases eliminate, the garbage created by control-flow operations in structured programs, in a way that is simple and elegant and mirrors the time-symmetry of the underlying physics.

[Note: After this memo was written, I noticed that many of the same ideas were previously developed in a paper by Hall, published in 1994 [3]. Thus, the present memo needs to be rewritten to credit Hall's contributions where appropriate.]

* "M" is for "Michael."

1 Introduction

A central problem of reversible computing is how to deal with the extra "garbage" bits of information that seem to inevitably result from useful computations. [1] A traditional computer simply destroys these bits, dissipating their energy into heat, but in sufficiently power- or cooling-limited environments, resorting to dissipation is undesirable, and it is the aim of our project to explore the alternatives.

Garbage bits can of course simply be stored until the end of a computation, at which point it may be collected by running the computation in reverse (after copying the output), so that the net garbage consists only of the information originally input to the computation. However, this does not address the large amount of garbage storage space that may be *temporarily* needed during the computation. Charles Bennett has shown [1, 2, 4] that the amount of temporary storage needed for garbage can be reduced further by reversing pieces of a computation in clever patterns. But so far as is known, linear-time algorithms can only do polynomial garbage reductions. Logarithmic garbage reduction can be achieved, but only at the cost of a polynomial increase in running time, which may be unacceptable for long-running programs.

Therefore, it is clearly in the interest of the designers of a reversible computing system (consisting of the processor architecture and one or more high-level language compilers) to minimize the amount of garbage that is produced in the first place.

Some garbage results from the execution of what Vieri has called *expanding* operations [5]: primitive reversible operations in which the number of interesting (non-constant) output bits exceeds the number of interesting input bits. No one knows for sure whether expanding operations are absulutely necessary for universal computing. Perhaps all programs could be rewritten in a form where only non-expanding operations are used, but this conjecture is unsubstantiated. So we are currently constrained to produce some garbage merely in the course of performing common operations such as AND and OR. There is not much that the architecture or compiler can do to avoid garbage creation in the face of requests for such operations from the programmer, except wait until enough of it has accumulated so that undoing a chunk of the computation and storing its input and output instead of the garbage trace becomes preferable, as is done in Bennett's algorithms.

Another potential kind of garbage, which is the subject of this memo, is that which results from arbitrary control-flow operations. In the worst

[1] Or, more precisely, from useful computations on arbitrary-sized inputs. For any function of bounded input size, there is a custom reversible logic circuit that computes the function using no more bits at any point than the number in the input plus the number in the output plus one [refs to Toffoli, Bennett, others?]. However, the construction may demand exponentially-sized circuits, and anyway, we are interested in general-purpose reversible computers without fixed limits on their input size. For such machines, asymptotic space requirements may be greater for reversible machines than for their irreversible counterparts when solving problems of interest. (However, this conjecture is unproven.)

case, imagine that every location in instruction memory contains a branch to location x. Then after executing one such instruction, we need to store as many bits as there are in instruction memory addresses to remember which previous instruction we just came from.

This observation leads to the simplest policy for handling control-flow operations: allow arbitrary branch and jump operations, but always push the address prior to the branch onto a special branch stack. This is the approach taken by Vieri [5] in the Pendulum architecture. No further modifications of traditional control flow operations are required. Vieri suggests placing a special COME-FROM instruction at the destination of every branch to make reverse operation easier to implement, but this is not strictly necessary; instead, the top of the branch stack can be continuously examined to see if the current address is referred to by the branch recorded on top of the stack, and if so, the reverse branch may be taken (on the assumption that control flow is never allowed to wrap circularly completely around the entire instruction address space).

Although simple, this approach has the obvious flaw, pointed out by Vieri, that it leads to a monotonically-increasing storage requirement for the PC garbage stack during the execution of any program with branches (which means any real program). Storage requirements monotonically decrease when running backwards, and so the *net* PC garbage storage for any program can be zero if programs are always reversed after execution, but this is of no help if the storage capacity of the system available for temporary storage of garbage data is limited to an amount that is too small to run the desired program.

This problem can be alleviated somewhat by making the previous address for the branch into programmer-accessible garbage, which can then be erased in software if its value can be reconstructed. This can be accomplished by having the branch place its source and destination addresses into a special "branch register;" other instructions can then swap the value of the register into a memory location, after which other code can reconstruct whether a branch was taken, and from where, using other means. This information can then be used to reconstruct and zero out the memory containing the stored previous value of the branch register. However, this approach inelegantly turns every branch into a somewhat clumsy jumble of garbage-cleanup code. Is there a way to avoid creating the branch garbage in the first place?

We observe here that whole "save the prior PC" approach to making control flow reversible is highly *time*-a*symmetric*. This is because the actual branch instruction, which contains all the information about the branch, is located at the source of the branch. At the destination of the branch, traditional programming styles assume that a special branch is not needed. But observe that the original pendulum architecture stipulates a COME-FROM statement at the destination of the branch, and my branch-register suggestion involves branch-register cleanup code at the destination of the branch. Thus we have already encountered the suggestion that there be *something* at the destination of a branch in a reversible architecture.

What if this something were, symmetrically, another branch instruction for use when operating in the reverse direction?

At first this suggestion seems insufficient. After all, traditional machine architectures allow that programs may contain branches from anywhere to anywhere based on any branch condition, so at a given point in a program there may be no simple condition that suffices to determine whether to branch when hitting a branch target in reverse. Worse, if many branches target a single location, a single instruction seems insufficient to allow branching back to any of them. These issues were observed and considered in more detail by Vieri [5].

But let us set aside, for the moment, the traditional architecture's ability to handle arbitrary branches, and instead consider the sorts of control flow structures that appear in actual structured programs. It is well known that three control-flow constructs suffice for general-purpose programming: `if-then-else` blocks, loops, and procedure calls. (Furthermore, loops and procedure calls are not both needed for universal computation, but it is convenient to have both available.) Automatic translation of programs that use additional constructs (such as C's `break`, `return`, `switch`, and `goto`) into programs using only these three constructs is generally straightforward and involves only modest overheads. [Need to read up on how to do it.] [ref: Dijkstra, or Boehm et al]

It turns out that, given a reversible architecture that supports symmetric forward and reverse branch instructions, all three basic constructs can be implemented in a simple, elegant way. No control flow garbage is produced, except by some loops, which in the worst case only produce $log_2 n$ bits of garbage, where n is the number of iterations of the loop. The legality of programs in this architecture is more restricted than in traditional architectures, in that the forward and reverse branch constructs must always be used only in a paired, symmetric way in order for energy dissipation to be avoided, but it is a simple job for compilers to translate an arbitrary program into symmetric machine code that satisfies this requirement. [Again, need to verify this.]

However, this issue makes compiler correctness even more vital than in traditional computers, as a compiler bug in this architecture could result in code that burns up a reversible processor implementation that is not designed to tolerate the heat dissipation. Automatic detection of illegal code sequences by the processor to avoid burn-up is an important area for future study. Possibly it can be solved simply by having the processor detect whenever an old PC value is reset using a value that is not equal to it, and thereupon raise an exception or halt. However, this idea needs more detailed elucidation at the register-transfer level.

2 Symmetric Control Flow Instructions

The fundamental feature of control flow instructions in this architecture is that each comes in two flavors: a "forward" version that is only used when the processor is running forwards, and a "reverse" version that is only used when the processor is running backwards. Each control-flow instruction is ignored (treated as a No-Op) when encountered during execution in the inappropriate direction. In the discussion here, reverse instructions are distinguished by prefixing them with a "*" character.

Control flow instructions come in four basic classes which are very traditional:

1. Conditional branches, which test some condition on some register or registers and set the program counter to point to a given location if the condition is satisfied.
2. Unconditional branches and jumps, which set the program counter to a given location unconditionally.
3. A jump-and-link or similar instruction for implementing procedure calls.
4. A jump-register or similar instruction for returning from procedure calls.

Each instruction in each of these classes comes in both forward and reverse flavors.

2.1 Example Instruction Specifications

Here are some example specifications for instructions in each of these classes. These examples are used later to illustrate how to implement each of the three basic structured control flow constructs (ifs, loops, and calls). This list is not intended to be comprehensive, since in a real architecture, we might want to support a wider variety of branch conditions, addressing modes, etc. Also some instructions' semantics might need to be changed slightly to make implementation easier. But the basic behavior of the instructions will be unchanged.

Each instruction implicitly exists in both forward and reverse flavors. Reverse flavors are distinguished by prefixing the instruction name with "*". All control flow instructions are defined to be No-Ops when executed in the wrong direction for their flavor, and the instruction descriptions describe only the operation of the instruction when encountered in the right direction.

Various instruction descriptions express conditions that must be satisfied. If these conditions do not hold, the processor will raise an exception, halt, or silently dissipate energy, depending on further design decisions. (Whichever it is, it probably ought to be consistent across all instructions.)

In general, the requirement is that every instruction in the entire architecture must cause the PC to change to point to an instruction which, if the processor's direction were to be immediately reversed, would return the PC exactly back to the current instruction. Later we will describe how to implement the 3 basic control flow constructs in a way that ensures this.

`BZ`,`BNZ` — Branch if equal to zero, branch if nonzero.

Tests a given register to see if it is equal (for `BZ`, not equal for `BNZ`) to zero, and if so, branches to the given immediately-specified relative offset from the current location. The destination address (if it is not the very next instruction) *must* contain a branch or jump of the opposite flavor that must be guaranteed to jump back to the current instruction if the processor were immediately to be reversed.

`BRA` — Branch always.

As above, but unconditional.

`JAL` — Jump and link.

Jumps to a given address, after placing the address of the next instruction after (or before, for the reverse flavor) the current one in a given register, which must have previously been ensured to contain zero. The destination instruction must be a `JR` instruction of opposite flavor.

`JR` — Jump Register.

Swap the PC with the value stored in the given register, which must be the address of a `JAL` instruction of opposite flavor that points to the current instruction. Then, reset the register to 0 using the pointer to the current instruction that is embedded in the target `JAL` instruction.

Note that, in addition to the obvious use for returning from procedure calls, `JR` can actually be used to implement run-time determined jumps to arbitrary locations, so long as those locations contain oppositely-flavored `JAL`s pointing back to the particular `JR`. Bizarrely, this behavior can be used to implement switch statements in addition to the intended use for procedure calls! [Remember to describe this in later section.]

The above are actually all the control flow instructions that are strictly necessary in a reversible architecture that supports ifs, loops, calls, and switches. More varieties of branch conditions are desirable, but superfluous.

3 Symmetric Control Flow Structures

The following diagram (fig. 1) illustrates how various control flow structures can be implemented in terms of the example instructions defined above.

In the diagram, vertical arrangements of instructions are supposed to suggest a linear sequence of instructions in memory, out of which only the control-flow instructions are shown. Arguments of instructions are omitted. Instruction streams off to one side may potentially be located elsewhere in memory far from the main linear sequence. Curved arrows

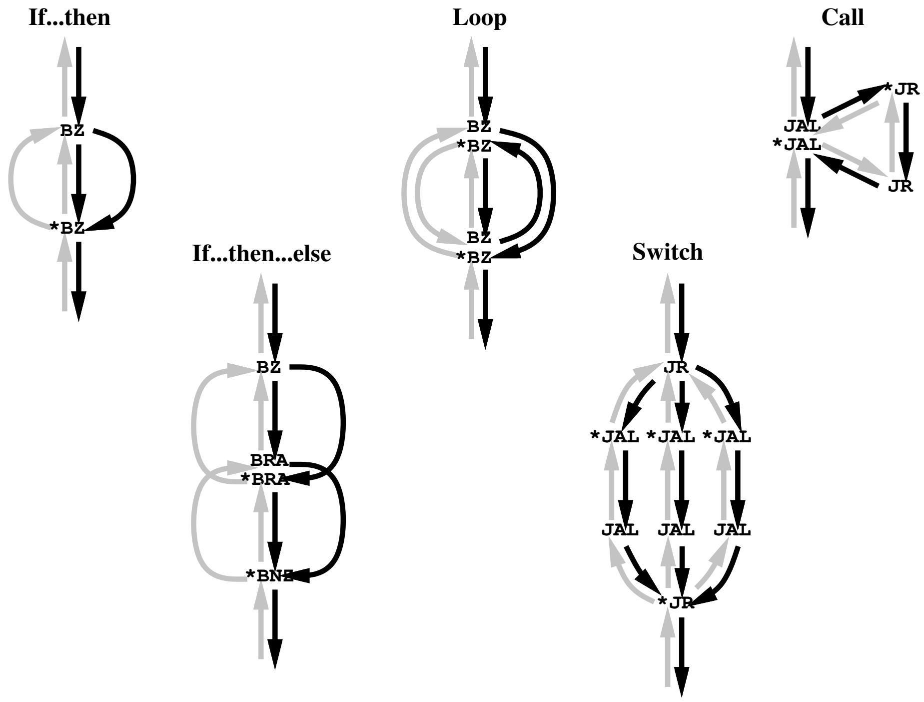

Figure 1: Examples of symmetric control flow structures.

(and some straight arrows) indicate branches and other nonlocal control flow. Black arrows indicate possible control-flow paths that may be taken when the processor is executing in the forward direction, while grey arrows indicate paths that may be taken when the processor is executing in the reverse direction.

Instructions prefixed by asterisks ("*") are reverse flavors which are ignored when encountered during forward execution, and vice-versa for instructions without asterisks.

3.1 Discussion of Structures

3.1.1 If...then (...else)

```
if (condition-expression) then
        body
end if
```

To compile an "if...then" construct into code, one simply places the result of the if's condition expression into a register that will remain unchanged during the body of the if, and then place identical but opposite-flavored conditional branches on that register at either end of the if body, pointing to each other. The value of the register may be uncomputed by extra code after the if statement, if desired.

Let us trace through the operation of the "if..then" in forwards execution. The code before the if is executed, evaluating the if's condition expression, and placing its result in a register, which, say, will contain 1 if the body is to be executed, 0 otherwise. Then the BZ statement is encountered. If the register is nonzero, the branch is not taken, and control flow proceeds into the body of the if. The body may of course temporarily swap the register value out to memory if the register is needed for other purposes, but if it does so, the original value must be swapped back in before the end of the body is reached. At the end of the body, the *BZ instruction is encountered, which does nothing. (Actually, at an implementation level, the processor may test the register again at this point to determine that the previously executed instruction was indeed the preceding instruction in memory, rather than the forward BZ, so that the prior PC value may be discarded properly. But this is not user-visible behavior.) Finally, after the whole "if", code may uncompute the value we conditioned on, or it might placed somewhere in memory for a future round of garbage cleanup. If the branch is taken, the body is skipped, and the *BZ at the destination of the branch tells the processor what the previous value of the PC was so that that previous value may be discarded.

Operation during reverse execution is exactly the same, except that the program counter normally proceeds backwards through the instruction stream, and the roles of forward and reverse instructions are exchanged.

[There is a problem with the policy of always reversing the condition expression after the "if" is finished, which is that if evaluating the condition expression it involves a recursive call of something involving an "if", then runtime can increase exponentially with the number of levels of these if calls, compared to not reversing the condition. The alternative is to store at least 1 bit of garbage information indicating which branch of the conditional was taken, and hope to recover this bit via some larger-scale reversal. However, if no recursive if's are involved in turning program variables into branch conditions, reversing the branch condition calculation is not a problem. Most if expressions don't involve recursive calls, so I think we're OK. Need to mention this issue more throughout text.]

```
if (condition-expression) then
        body1
else
        body2
end if
```

The "If...then...else" statement is very similar to the "if...then", except that the conditional branches go to opposite-flavored unconditional branch instructions. This does not prevent the previous PC value from being discarded properly.

3.1.2 Loops

Loops get a little bit tricky. Traditional structured programming loops (while, until, do-while) are generally associated with a condition that is true when in the loop, and false when the loop is about to end. But when running in reverse, what tells us that the loop is about to reach the beginning? The condition associated with the loop must change its value during *both* the first and the last iterations of the loop. (Or neither, if there is only 1 iteration.) It must of course change during the last iteration so that its value tells us that now it is time to leave the loop. But also, it must change during the first iteration so that when running in reverse, when we reverse all the way back to the first iteration, its change tells us that now it is time to go back out of the start of the loop. This leads to the following semantics for the loop condition:

> The loop condition is the condition for *either* exiting or entering the loop, depending on whether we are already in the loop or not. In other words, when the loop boundaries are encountered, either from inside the loop or from outside, the condition tells us whether to cross over to the other side, or remain on the side we're already on.

```
symloop (condition-expression)
       body
end symloop
```

We can imagine a new structured programming construct, "symloop," for "symmetric loop," similar to a traditional while loop, but with its condition interpreted as described above. In other words, if the condition is false when the loop is first encountered, the loop is not executed at all. If it is true the loop is entered, and if it remains true at the end of the first iteration, the loop is immediately exited, performing only one iteration. To execute more than one iteration, the condition must become false during the first iteration, and then become true again during the last iteration.

The symloop can be implemented as shown in figure 1. The structure is essentially equivalent to an if, but branching backwards instead of forwards, and embedded within a forwards-branching if testing the same condition. In the example, each `BZ-*BZ` sequence serves as one of the boundaries of the loop body. Whenever a boundary is encountered, whether from inside the loop, from outside the loop, running forwards, or running backwards, the boundary causes the appropriate thing to occur: control flow switches betweeen the loop's inside and outside if and only if the loop condition (in this case that some tested register is non-zero) is satisfied.

Programmers might be easily able to use the symloop construct directly to implement many kinds of loops (i.e., to loop an index i between the values min and max, the condition is simply (i=min or i=max+1)). However, arbitrary "while"-style loops can be supported as well via the simple expedient of introducing an extra "count" variable to keep track of the number of times through the loop. For example, a traditional while loop, such as the following,

```
while (continuation-condition-expression)
        body
end while
```

can be automatically transformed into the following symloop:

```
count = 0
symloop ( ((count = 0) and (continuation-expression-condition))
          or
          not (continuation-expression-condition)
        )
        body
        count = count + 1
end symloop
```

In forward operation, the first part of the disjunctive condition is the condition for entering the loop, and the second part is the condition for exiting the loop. In reverse, however, the second part becomes the condition for entering the loop in reverse, and the first part becomes the condition for exiting out the start of the loop after completing the first iteration.

Note, however, that this approach produces a count of the number of times through the loop as a garbage value. However, this is much better than generating another word of garbage on every loop iteration as would happen if loops simply used branches that pushed the prior PC onto a stack.

3.1.3 Procedure Calls

Procedure calls are straightforward; the call consists of forward and reverse `JAL`'s back-to-back that refer to the beginning and end of the procedure, respectively. At each end there is an appropriately-flavored `JR` to return control back to the caller.

One interesting thought is to have another version of JAL that reverses the processor direction as well as changing the PC. Then, a single subroutine can be called in both forward and reverse modes from a single caller running in forward mode. This reduces code size during garbage cleanup, since forward and reverse versions of the procedure do not both have to be provided.

3.1.4 Switch (Case) Statement

This is an interesting application of `JAL` and `JR` in a way other than for the procedure calls for which they were intended. Before the switch, the address of the branch to take is computed by evaluating some expression. We jump to the computed address, execute the body of the particular case, and then jump to the end of the statement storing the address of the *end* of the appropriate case in a register, which can then be uncomputed by code similar to the code that generated the address of the start of the appropriate case. The end code will need to be a bit different so that it uncomputes the end address rather than the start address of the branch taken, but this is trivial for a compiler.

4 Conclusion

We have shown that all the basic control-flow constructs of structured programming can be implemented in a totally time-symmetric way that produces no garbage except the occasional count of the number of iterations of a loop whose number of iterations cannot be predicted rather than by executing it.

This advancement holds the promise of greatly simplifying the problem of garbage accumulation in reversible computers, since it nearly eliminates the entire class of garbage that is associated with control flow operations, meaning that the only garbage that is produced is due to the application of "expanding" data operations.

Future research will hopefully show how us to reduce data garbage as well, although it probably cannot be eliminated entirely or even limited to, say, a linear function of non-garbage storage. In any case, appropriate use of subcomputation reversals, such as in Bennett's algorithms, should be sufficient for allowing fully reversible architectures to perform useful computations.

References

[1] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 6:525–532, 1973.

[2] C. H. Bennett. Time/space trade-offs for reversible computation. *Society for Industrial and Applied Mathematics Journal on Computing*, 18(4):766–776, 1989.

[3] J. Storrs Hall. A reversible instruction set architecture and algorithms. In *Physics and Computation*, pages 128–134, November 1994.

[4] Robert Y. Levine and Alan T. Sherman. A note on Bennett's time-space tradeoff for reversible computation. *Society for Industrial and Applied Mathematics Journal on Computing*, 19(4):673–677, 1990.

[5] Carlin J. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.